UMIACS-TR-90-51 April 1990
CS-TR -2450

A Software-System Visualization Tool

John R. Callahan
Department of Computer Science
University of Maryland
College Park, MD 20742

James M. Purtilo
Institute for Advanced Computer Studies and
Department of Computer Science
University of Maryland
College Park, MD 20742

COMPUTER SCIENCE TECHNICAL REPORT SERIES

UNIVERSITY OF MARYLAND
COLLEGE PARK, MARYLAND
20742

UMIACS-TR-90-51
CS-TR -2450

April 1990

A Software-System Visualization Tool

John R. Callahan
Department of Computer Science
University of Maryland
College Park, MD 20742

James M. Purtilo
Institute for Advanced Computer Studies and
Department of Computer Science
University of Maryland
College Park, MD 20742

Abstract

Minion is a visual link editor used to configure and instrument software systems. It was originally developed as a general-purpose graph editor for use in many task domains where connected graphs appropriately model logical structures. A software system, for example, can be represented by a connected graph — nodes and edges of the graph correspond respectively to large-grain software modules (e.g., programs, files, databases, servers) and the dependencies established between these modules (e.g., procedure calls, message passing channels). We use the tool to write, visualize, and execute specifications for heterogeneous, distributed software applications in the Polylith module interconnection environment. Our tool complements use of the Polylith textual module interconnection language (MIL) because Minion is used as a link editor that reads and writes MIL specifications. Through the use of windows, icons, menus, and other means of direct manipulation, users can edit MIL specs as graphs in order to perform complex configuration management tasks which may be tedious to denote directly in the MIL.

This research is supported by Honeywell and the DARPA/ISTO Common Prototyping Language initiative, with oversight provided by Office of Naval Research.

1 INTRODUCTION

This paper describes details of the MINION link editor, a visual configuration management tool developed for the POLYLITH module interconnection environment. In MINION, software systems are represented by connected graphs for purposes of visualization, construction, and instrumentation. Nodes and edges of a graph correspond respectively to large-grain software modules (e.g. programs, databases) and the dependencies established between modules (e.g., procedure calls. channels). Through the use of windows, icons, menus, and direct manipulations, MINION users can edit these graphs in order to perform complex configuration management tasks which may be tedious in textual approaches.

Module interconnection languages (MILs) have largely been textual and POLYLITH is no exception. MINION complements, rather than rivals, our textual approach. Either MINION or the POLYLITH MIL may be used to describe the structure of software systems. MINION, however, easily facilitates quick prototyping, visualization, and instrumentation tasks which would require extensive editing of POLYLITH MIL specifications. For example, we can graphically attach "listeners" to all edges of a graph in order to monitor message passing activities. Such a task would require tedious editing of all bindings expressed in a textual specification. The MINION link editor reads in textual POLYLITH descriptions of software systems and displays them as connected graphs. Users may then manipulate the components of graphs and generate new textual descriptions of the systems as needed. This complements use of the POLYLITH MIL as a textual representation of the structure of software systems. Thus, users are not limited to text editors when designing software systems. They may leverage the advantages of graphical representations against the complexity of large-grain configuration programming tasks.

In either case, it is our intent to expedite the construction and manipulation of the logical structure of software systems. Abstractly, program components, called modules. can be bound together into systems. In POLYLITH, these modules may be written in different programming languages and reside in shared or separate protection domains (i.e., file systems, memory spaces. machines). Module composition mechanisms must manage communication and control across these domain boundaries. A POLYLITH run-time system, called a *software bus*, abstractly represents the specific connections built from descriptions of particular systems of modules. For this reason, the POLYLITH MIL is an extensible, object-oriented language designed to accommodate the properties of many diverse programming languages and protection domains.

Configuration management tools are forced to operate in these diverse and dynamic environments. To this end, MINION is also a run-time entity. It may be used in conjunction with local debugging tools. One can monitor the state of software modules and their links to other components. For example, users can suspend execution of processes, examine the value of module variables, and monitor occurrences of local and remote procedure calls. Through the use of animation, users can visualize the paths of execution in a software system in order to identify problems, isolate bottlenecks, and test solutions. In this role, MINION has been valuable to us as a debugging and prototyping tool.

In this paper, we will examine some examples of visual configuration programming. First, we present more details of the POLYLITH module interconnection language and environment. Next, we use MINION in the visual construction and execution of an example software system. Finally, we discuss some of the implementation issues involved in developing MINION.

2 POLYLITH

Software components in POLYLITH are written in different programming languages and reside in different protection domains. They can be described abstractly, however, through the use of object specifications. Because POLYLITH is object-oriented, specifications need not list all of their properties explicitly, but can inherit some properties from their environments implicitly.

POLYLITH specifications come in two types: modules and systems. A module may be a description of almost anything — a file, program, database, or a set of procedures and functions. It is simply a software object that advertises a set of properties. A module's type is determined by the set of properties it contains. For example, program modules can contain procedural interface descriptions. Abstractly, an interface describes a provided or used resource (e.g., a set of exported and imported procedures). A program module may also contain information about its implementation characteristics: a source file name, object file name, binary file name, symbol table, compiler, linker, and machine location. Default values for some properties are bound late in the configuration process or during execution through inheritance and parameterization mechanisms.

A system description creates, binds, and parameterizes instances of module descriptions together into a single application that may cross language boundaries and protection domains. Figures 1 and 2 display program module descriptions and a system description for the Dining Philosophers problem [Dijk65] implemented in POLYLITH. In the Dining Philosophers problem, we place n diners and n forks alternately around a table with a bowl of rice in the center. Each diner must use two adjacent forks to eat. Once finished eating, a philosopher release the forks, thinks for a period of time, and repeats the cycle. We must implement a system of such processes such that there is no deadlock (i.e., a stalemate over forks) and that all diners will eventually eat. In this example, the system description creates three instances of each module type and binds them explicitly by linking source and sink interfaces. Each diner instance represents the start of a separate thread of control because it contains a main interface. Fork instances do not contain main interfaces and act as servers that accommodate a single thread of control at a time (the default for program modules with no main interface).

The three POLYLITH specifications are configured into a single description that when executed constructs the connections necessary to implement the system in a particular execution environment. If the appropriate components are accessible, a software bus is built and the system begins execution. A software bus is an abstraction that represents a collection of message-passing stubs, local and remote procedure call stubs, and communication channels. Components may be

```
module "Diner" : {
        implementation : {
                machine : "flubber.cs.umd.edu"
                source : "diner.c"
        }
        main : true
        source "reqRght" : { }
        sink   "getRght" : { }
        source "reqLeft" : { }
        sink   "getLeft" : { }
        source "relRght" : { }
        source "relLeft" : { }
        client "printf"  : { string ;
               { integer } { string } }
               => { integer }
}
```

```
module "Fork" : {
        implementation : {
                machine : "brillig.umd.edu"
                source : "fork.p"
        }
        main : false
        multisink   "listen"  : { }
        multisource "grant"   : { }
        multisink   "release" : { }
        client      "writeln" : { string }
                    => { }
}
```

Figure 1: POLYLITH module specifications for the Dining Philosophers problem.

```
system "Dining Philosophers" : {
        tool "Diner 1" : "Diner"
        tool "Diner 2" : "Diner"
        tool "Diner 3" : "Diner"
        tool "Fork 1" : "Fork"
        tool "Fork 2" : "Fork"
        tool "Fork 3" : "Fork"

        bind "Diner 1" "reqLeft"    to "Fork 1" "listen"
        bind "Diner 1" "getLeft"    to "Fork 1" "grant"
        bind "Diner 1" "relLeft"    to "Fork 1" "release
        bind "Diner 1" "reqRght"    to "Fork 3" "listen"
        bind "Diner 1" "getRght"    to "Fork 3" "grant"
        bind "Diner 1" "relRght"    to "Fork 3" "release"
        bind "Diner 2" "reqLeft"    to "Fork 2" "listen"
        bind "Diner 2" "getLeft"    to "Fork 2" "grant"
        bind "Diner 2" "relLeft"    to "Fork 2" "release
        bind "Diner 2" "reqRght"    to "Fork 1" "listen"
        bind "Diner 2" "getRght"    to "Fork 1" "grant"
        bind "Diner 2" "relRght"    to "Fork 1" "release"
        bind "Diner 3" "reqLeft"    to "Fork 3" "listen"
        bind "Diner 3" "getLeft"    to "Fork 3" "grant"
        bind "Diner 3" "relLeft"    to "Fork 3" "release
        bind "Diner 3" "reqRght"    to "Fork 2" "listen"
        bind "Diner 3" "getRght"    to "Fork 2" "grant"
        bind "Diner 3" "relRght"    to "Fork 2" "release"
}
```

Figure 2: A POLYLITH system specification for the Dining Philosophers problem.

connected together directly via whatever communication channels are common to connected components (e.g., TCP sockets) or they may communicate through a common, centralized message handler. The process of building a software bus is complex and is covered in detail in [Purt86].

3 CONSTRUCTION

We shall construct a connected graph using MINION that will correspond to the system specification in Figure 2 using diner and fork module specifications. We will assume that these are made available from POLYLITH libraries and symbol extraction tools [PuCa89]. Nodes and edges of the graph will represent instances and bindings respectively. Figure 3 illustrates the first step of this process. The diner and fork specifications are listed as modules in the MINION palate. The palate holds descriptions for modules listed in the user's file system (e.g., a current directory). Components may also be found in module libraries. In Figure 4, the user copies these onto the MINION viewport as instances. In this case, instances are numbered, but the user can attach any name to an instance or parameterize MINION with procedures for generating instance names.

Next, the user creates links between modules by joining the appropriate interfaces. Specific interfaces of different modules are linked individually by drawing edges from one instance to another. In order to select the interface to be bound, each instance has a disposition: it is either active or inactive. One can change a node's disposition by selecting an interface from its instance menu. If any of the interfaces are chosen, then the node is activated. An active node is said to be available on one of its interfaces. Figure 5 illustrates the use of an interface menu. One of the fork instances (Fork 1) is currently active on its **grant** interface, while the disposition the diner instance (Diner 1) is being changed. The other fork instance (Fork 2) is inactive. An instance changes its iconic labeling to reflect its "current" disposition[1]

Links are created in many ways in MINION. One can select a node by pressing a mouse button over it, dragging the mouse, and releasing the button once over another node. The two nodes may be active or inactive. If they are active, this called an *interface join* because two specific interfaces are joined together explicitly. After the link is complete, the two instances become inactive. If one or both nodes are inactive, unbound interfaces are joined by name and type. This is called an ***unspecified join*** because it is implicit. A prompt window is displayed which asks the user to verify the implicit bindings. This feature may be disabled in order to bind whole systems silently. In addition to these node-to-node linking mechanisms, MINION permits selection of groups of nodes to be bound implicitly by name and type. In our example, the diners and forks do not share interface names and their types are void because no arguments are passed within messages.

Implicit bindings are also a part of the textual POLYLITH MIL through the use of grammatical scoping mechanisms and inheritance, but they remain conceptually transparent to the designer.

[1] In later versions of MINION, arbitrary PostScript code will be used to describe iconic representations of dispositions.

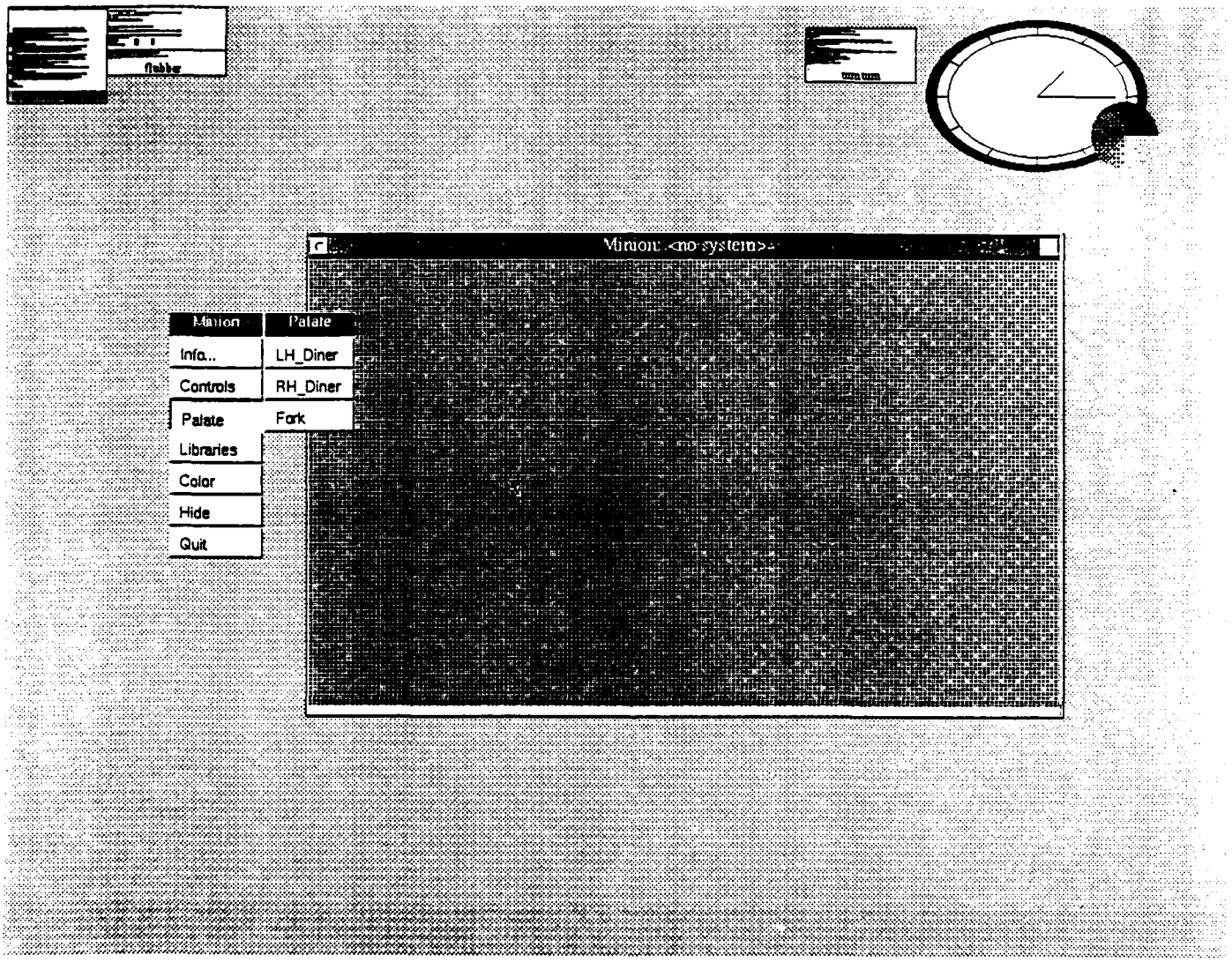

Figure 3: An empty MINION viewport with attached frame and palate menus.

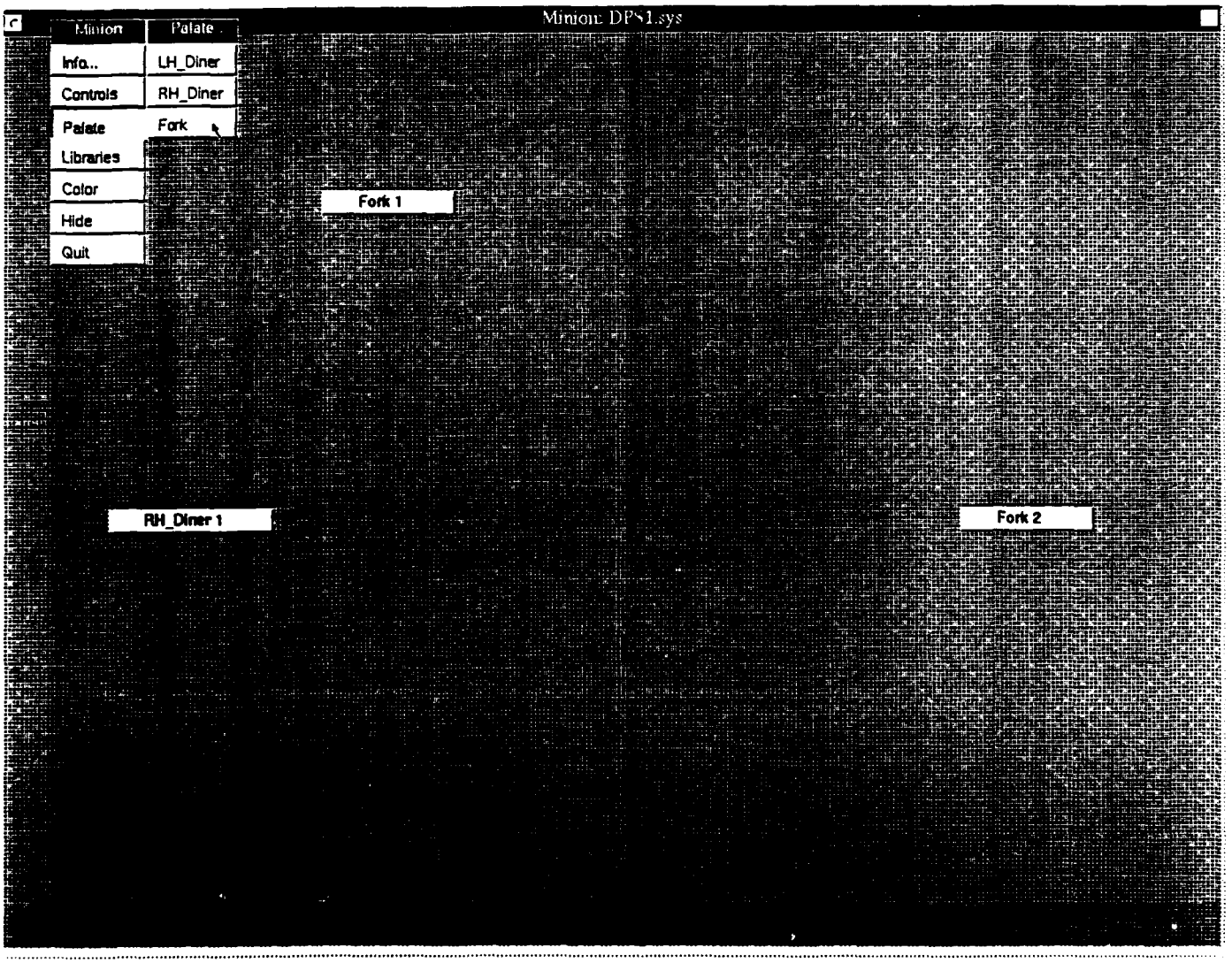

Figure 4: Three instances copied from the palate to the MINION viewport.

An implicit binding may be incorrect (e.g., a misspelling) but undetectable prior to compilation. A MINION user can view these implicitly constructed connections and notice incorrect links to library modules and "regions" of a graph. We can scale and reconfigure graphs interactively and selectively view subgraphs of a system in order to isolate binding problems. Consider the graph in Figure 6. We can compare it with an equivalent textual description in Figure 7. The error seems much more obvious from viewing the graph because we know that solutions to the Dining Philosophers problem are structurally cyclic in general.

Figure 8 is a snapshot of the completed graph. Each link represents three connections (i.e., request, get, release). Links can be viewed in two ways: through the link or a node. By clicking on the link, the connected node labels will cycle through the connections, i.e., the node labels and arrows on different edges will change with each click of a mouse button on the link. This has been done in Figure 8. An arrow is displayed showing the direction of the message channel and the connected nodes display the names of the relevant interfaces. Alternatively, one can select an already bound interface from an instance menu and view the existence of a particular connection. Links between two modules can be split and joined in order to present a system in the most graphically pleasing manner.

4 EXECUTION

We can now build an executable image of our system and run it. During execution, the MINION tool is iconified and windows are created for each instance as specified by their module descriptions. By default, each instance gets a window that represents its standard input and output channels. For example, diner instances have a window tool associated with them, but fork instances simply print a log of their activities to their standard output channels. The diner windows have four labels that highlight to show the state of the program — requesting, eating, releasing, or thinking. Normally, the default windows are not displayed. Figure 9 illustrates execution of the three philosopher example. Three customized windows are created which graphically display the activities of each philosopher.

In the bottom left corner of the screen in Figure 9, the designer uses the icon menu to get a default window for one of the forks. Once selected, this window appears and shows a log of fork states as printed on its standard output channel. Default windows can also be used to debug modules. We can suspend execution and examine the state of module variables using local debugging tools.

During execution, the MINION viewport visually depicts message passing activities by labeling nodes and shading edges. The user can speed up, slow down, or step through message passing activities. In Figure 9, we notice that the system is in a deadlocked state — all diners windows are labeled "Requesting ...". In Figure 10, we show the MINION viewport also in a state of deadlock. During execution, dark directed edges represent messages sent but unreceived. Light directed edges denote recently sent and received messages, i.e., the last message sent. All three of the diner instances are labeled "RH_Diner X: getLeft" because that was the last interface used by

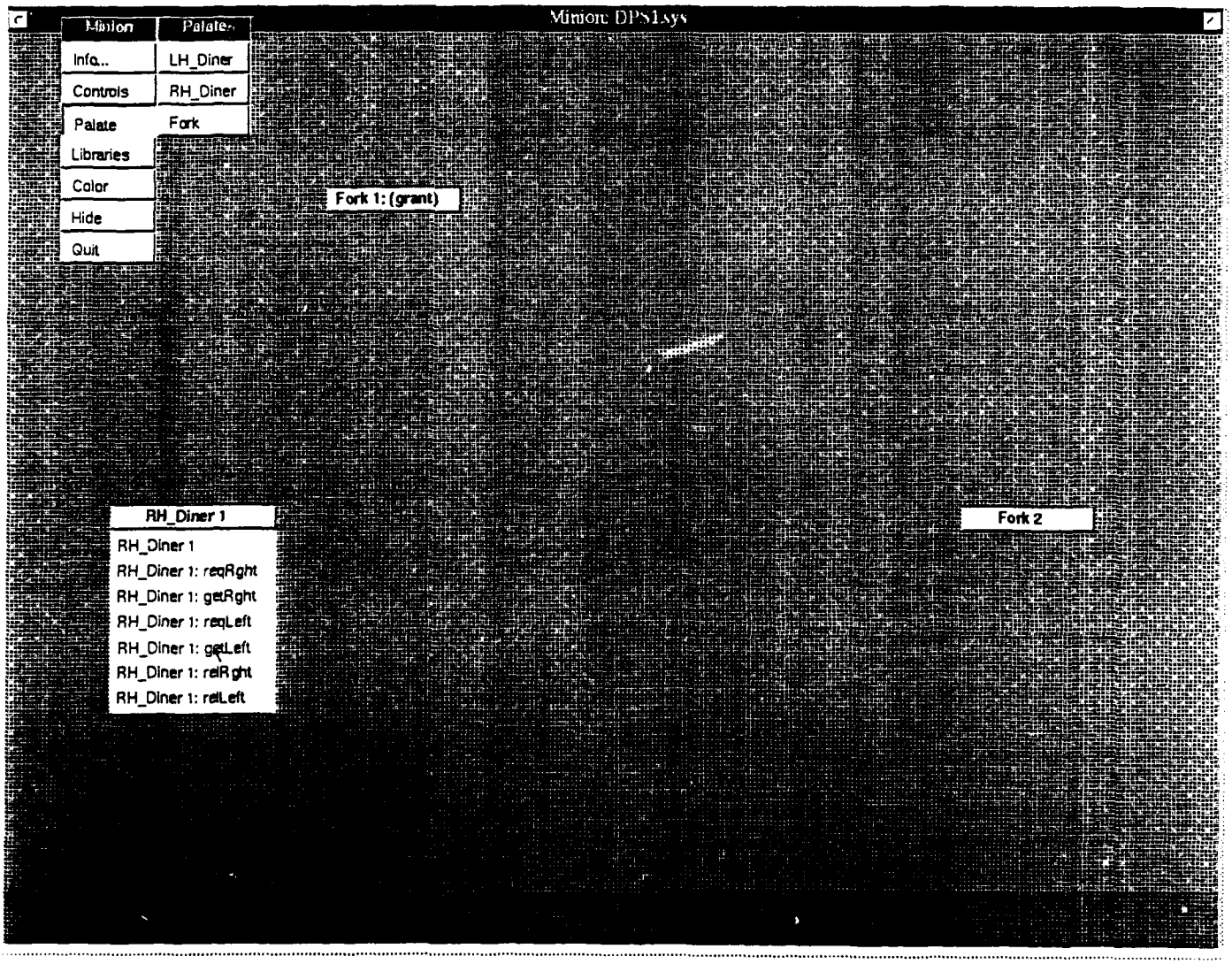

Figure 5: Changing an instance's disposition.

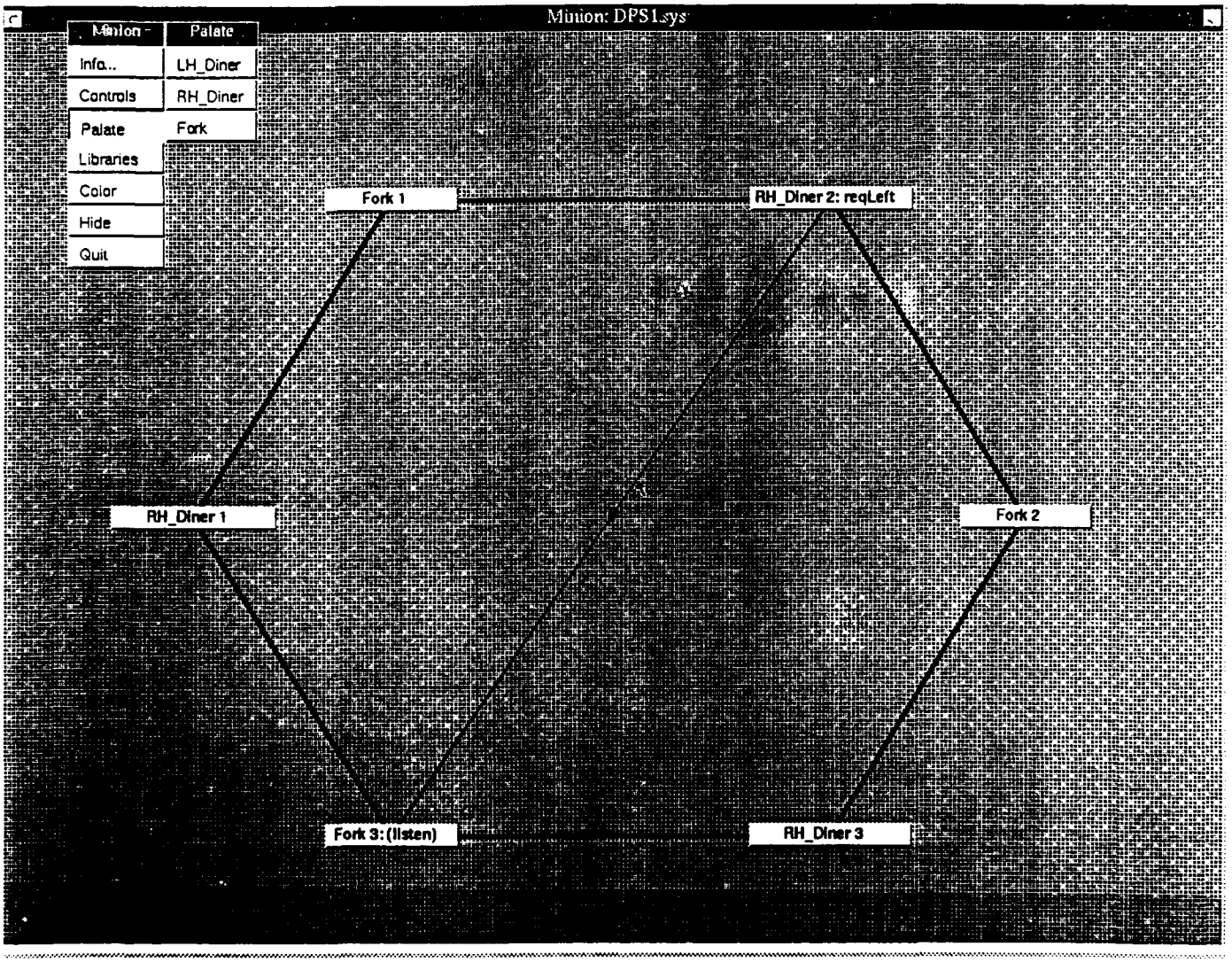

Figure 6: Visualization of a connection error.

```
system "Dining Philosophers" : {
        tool "Diner 1" : "Diner"
        tool "Diner 2" : "Diner"
        tool "Diner 3" : "Diner"
        tool "Fork 1" : "Fork"
        tool "Fork 2" : "Fork"
        tool "Fork 3" : "Fork"

        bind "Diner 1" "reqLeft"        to "Fork 1" "listen"
        bind "Diner 1" "getLeft"        to "Fork 1" "grant"
        bind "Diner 1" "relLeft"        to "Fork 1" "release
        bind "Diner 1" "reqRght"        to "Fork 3" "listen"
        bind "Diner 1" "getRght"        to "Fork 3" "grant"
        bind "Diner 1" "relRght"        to "Fork 3" "release"
        bind "Diner 2" "reqLeft"        to "Fork 3" "listen"
        bind "Diner 2" "getLeft"        to "Fork 2" "grant"
        bind "Diner 2" "relLeft"        to "Fork 2" "release
        bind "Diner 2" "reqRght"        to "Fork 1" "listen"
        bind "Diner 2" "getRght"        to "Fork 1" "grant"
        bind "Diner 2" "relRght"        to "Fork 1" "release"
        bind "Diner 3" "reqLeft"        to "Fork 3" "listen"
        bind "Diner 3" "getLeft"        to "Fork 3" "grant"
        bind "Diner 3" "relLeft"        to "Fork 3" "release
        bind "Diner 3" "reqRght"        to "Fork 2" "listen"
        bind "Diner 3" "getRght"        to "Fork 2" "grant"
        bind "Diner 3" "relRght"        to "Fork 2" "release"
}
```

Figure 7: A POLYLITH system specification with a connection error.

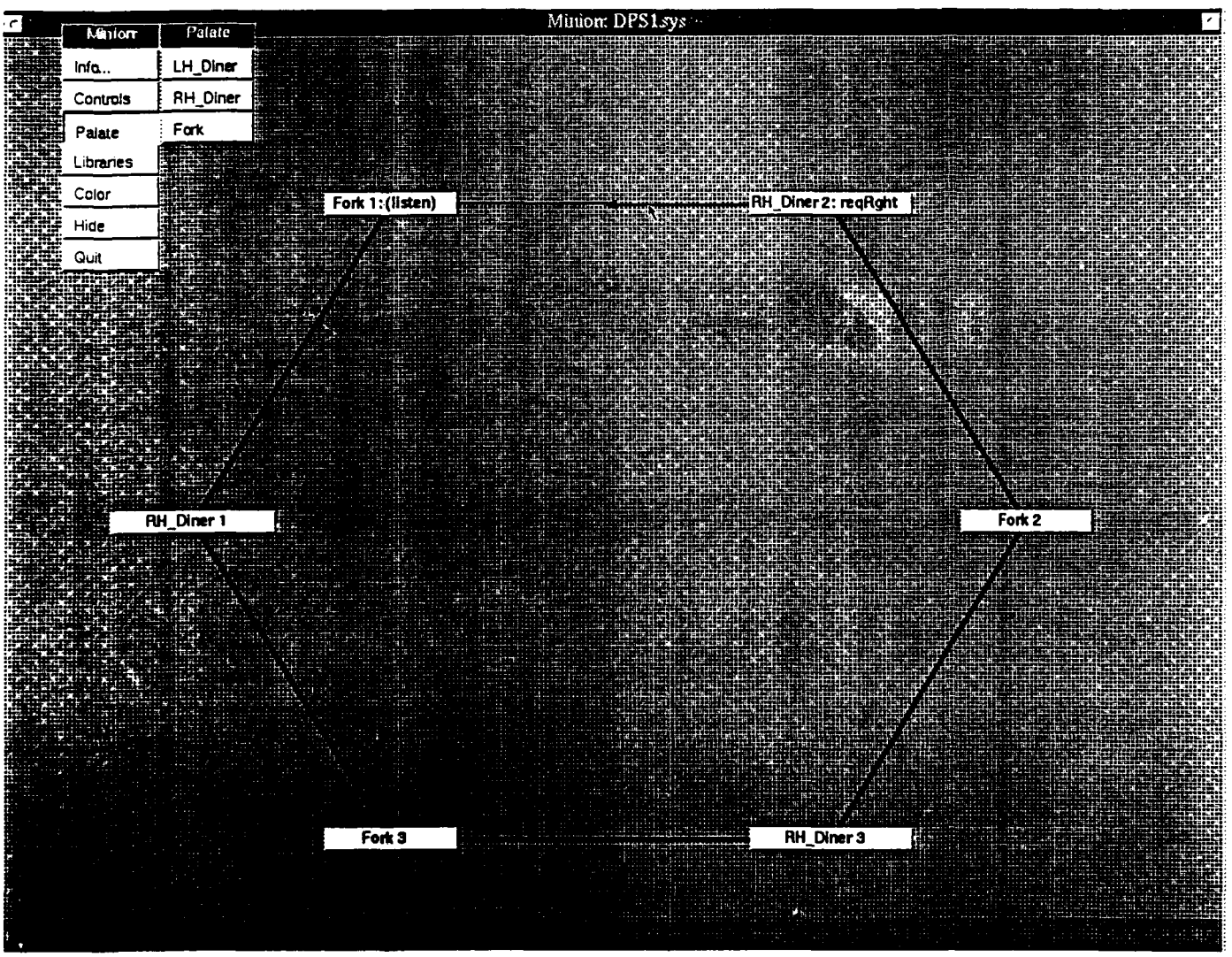

Figure 8: A graphical (but incorrect) solution to the Dining Philosopher's problem.

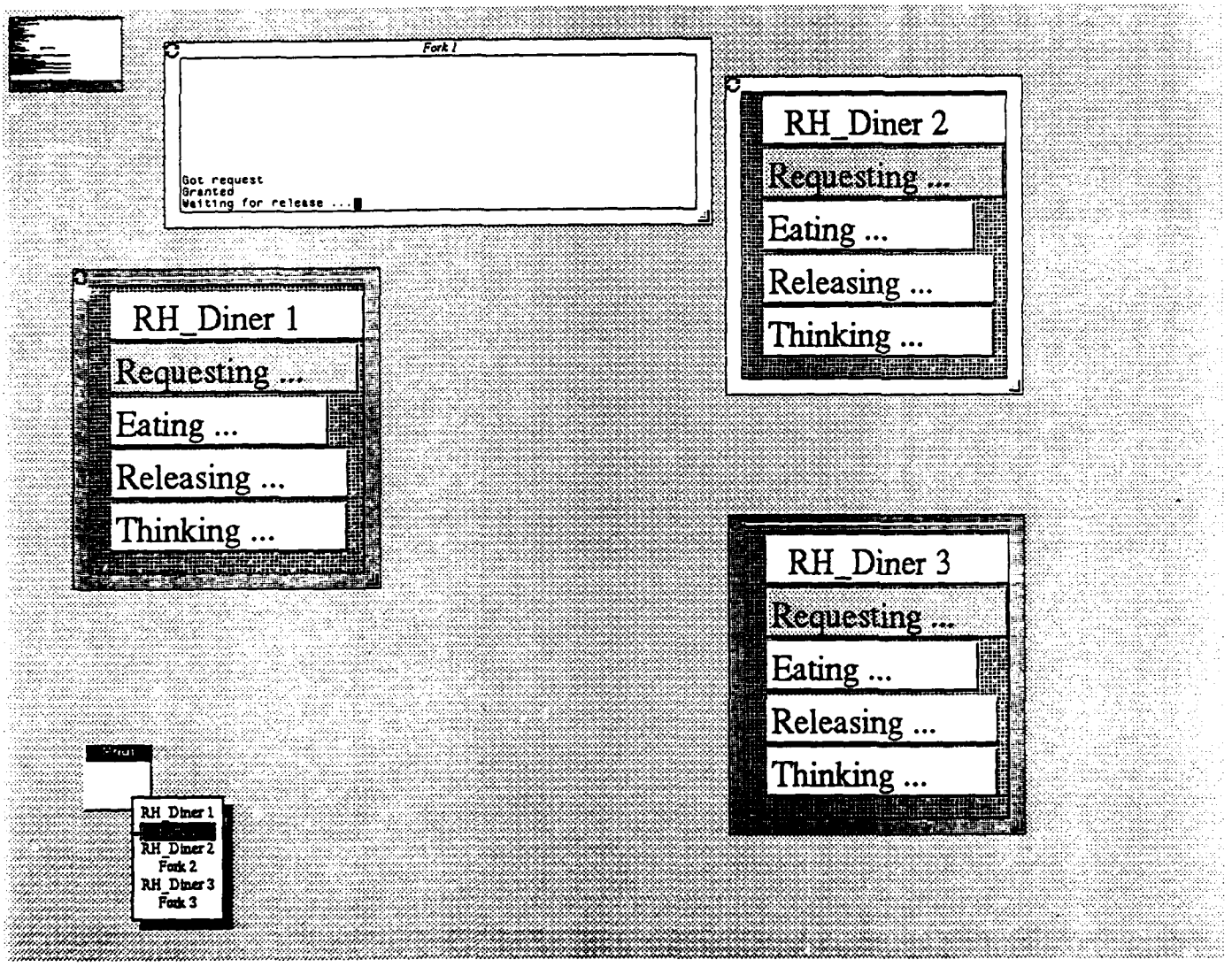

Figure 9: Opening a default window.

the tool. All three fork instances are labeled "Fork *X*: granted" because all `getRght` calls were honored. This is the source of the problem. The diners are all *right-handed.* All diners requested and received their right forks first, putting the system into a deadlocked state. Any solution to the dining philosophers problem must be acyclic in order to prevent deadlocks. We can delete one of the nodes and substitute a new instance of a left-handed philosopher in its place from the library. The corrected solution system is renamed and shown in Figure 11.

5 DISCUSSION

Configuration management systems tend to suffer from clutter problems created by the myriad of module dependencies. This is especially true in large software systems and presents a problem to visual configuration programming systems. MINION handles this problem by hiding unimportant details or scaling them graphically. We can graphically select subgraphs of graphs and compress them into nodes. A node that represents a compressed graph is simply a system description. Its interface menu is a composite list of all internally unbound interfaces. For example, if we were to compress the whole graph in Figure 11, the resulting system would have no internally unbound interfaces because it is a complete system. Thus, its interface menu would be empty.

Libraries, on the other hand, are incomplete systems represented by compressed graphs. For example, in the Dining Philosophers example, not all of the interfaces were explicitly bound. The remaining interfaces (`printf` and `writeln`) were bound implicitly to library routines linked directly into the executables of each instance. Recall that the links between modules need not be message passing channels. The POLYLITH software bus makes binding decisions and generates stubs based on protection domain properties. For example, links to local library modules are simply dependency links that use local binding mechanisms.

The library nodes in Figures 3-11 are not visible in the viewport because they have been made transparent. Compressed graphs may become transparent in order to reduce clutter. All links to transparent nodes also become transparent. One of the enhancements we plan for MINION is the ability to compress or make transparent all nodes with particular properties automatically.

By reducing visual clutter, we increase our capacity to abstract. We can focus our attention on crucial details instead of *all* details. This is the case for link editors in more tightly-coupled environments — invocations are implicitly bound by name. In MINION, binding mechanisms are similarly transparent. For example, new modules can be spliced *onto* links. This is done by moving or copying an instance of a module onto a link. The module must have a *listener* property in order to be merged into an existing connection. For instance, we may wish to monitor message passing activities in our Dining Philosophers system. It is a relatively simple task to instrument each edge with a listener module. It would take six copies from the library palate; one listener for each edge of the graph. On the other hand, Figure 12 illustrates the complexity of the textual consequences of the rebindings.

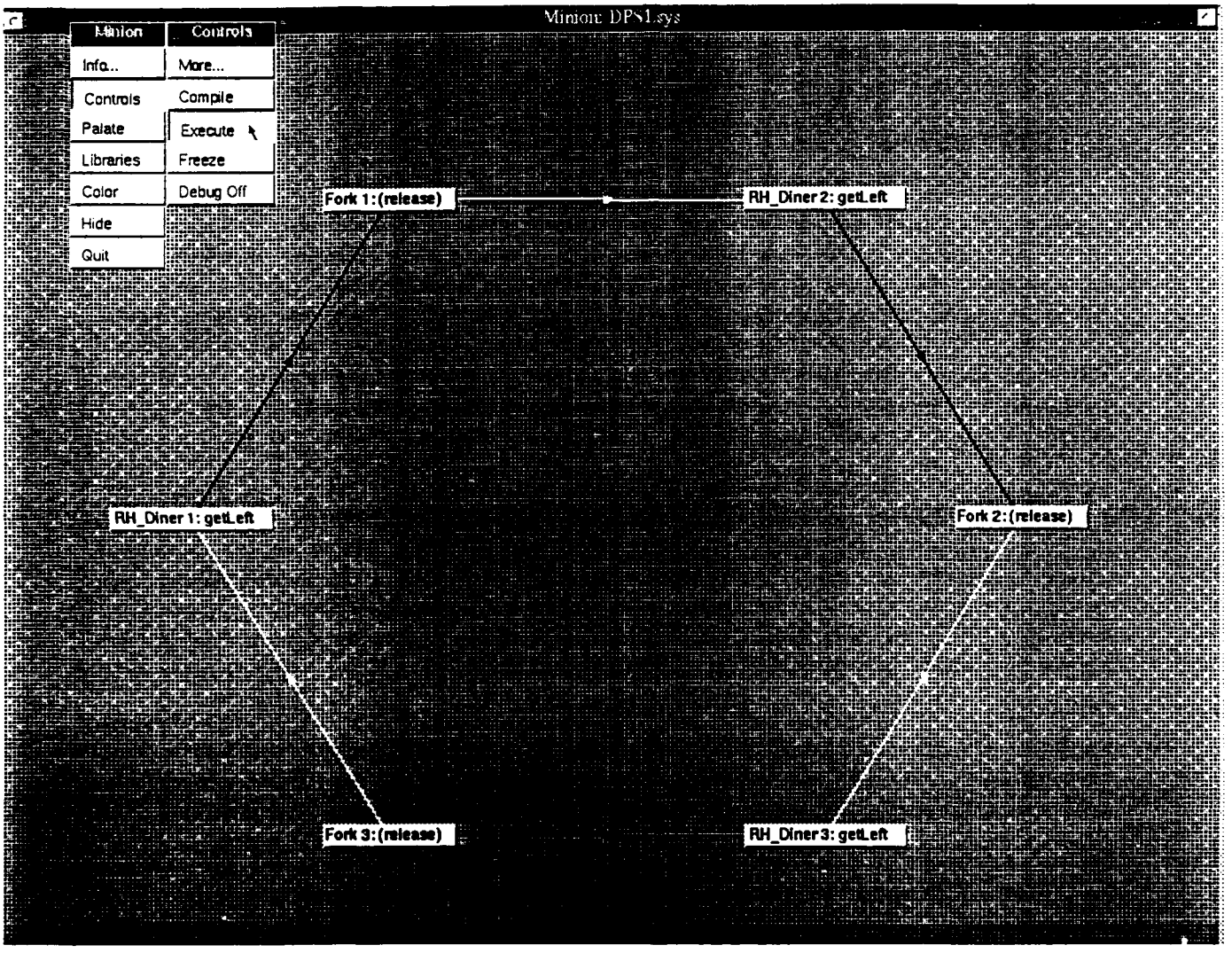

Figure 10: Evidence of deadlock in message patterns.

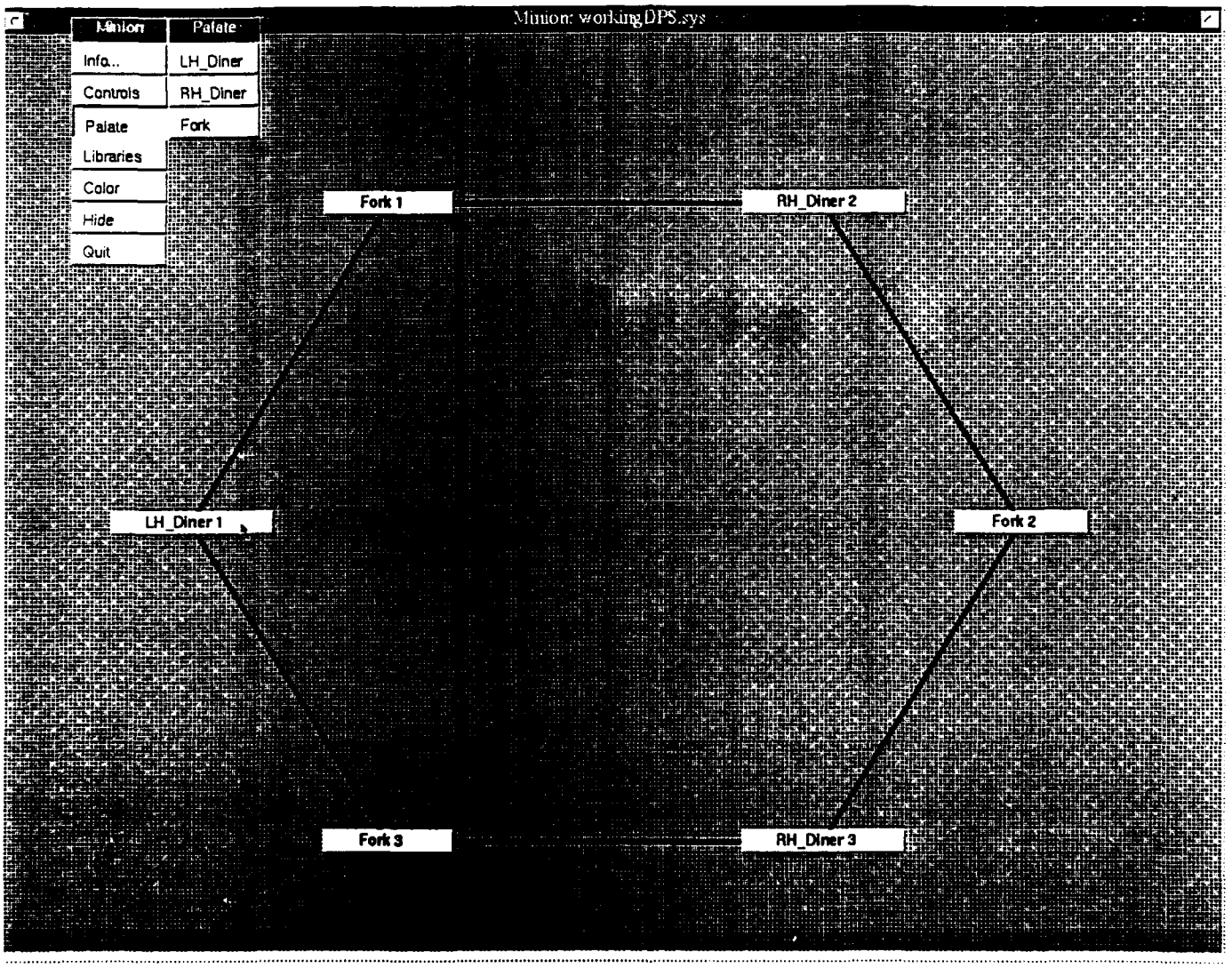

Figure 11: A corrected solution to the Dining Philosopher's problem.

```
system "Dining Philosophers" : {
        tool "Diner 1" : "Diner"
        tool "Diner 2" : "Diner"
        tool "Diner 3" : "Diner"
        tool "Fork 1" : "Fork"
        tool "Fork 2" : "Fork"
        tool "Fork 3" : "Fork"
        tool "L1" : "Listener"
        tool "L2" : "Listener"
        tool "L3" : "Listener"
        tool "L4" : "Listener"
        tool "L5" : "Listener"
        tool "L6" : "Listener"

        bind "Diner 1" "reqLeft"          to "L1" "reqIn"
        bind "L1" "reqOut"                to "Fork 1" "listen"
        bind "Diner 1" "getLeft" to "L1" "getOut"
        bind "L1" "getIn"                 to "Fork 1" "grant"
        bind "Diner 1" "relLeft" to "L1" "relIn"
        bind "L1" "relOut"                to "Fork 1" "release
        bind "Diner 1" "reqRght" to "L6" "reqIn"
        bind "L6" "reqOut"                to "Fork 3" "listen"
        bind "Diner 1" "getRght" to "L6" "getOut"
        bind "L6" "getIn"                 to "Fork 3" "grant"
        bind "Diner 1" "relRght" to "L6" "relIn"
        bind "L6" "relOut"                to "Fork 3" "release"
        bind "Diner 2" "reqLeft"          to "L3" "reqIn"
        bind "L3" "reqOut"                to "Fork 2" "listen"
        bind "Diner 2" "getLeft" to "L3" "getOut"
        bind "L3" "getIn"                 to "Fork 2" "grant"
        bind "Diner 2" "relLeft" to "L3" "relIn"
        bind "L3" "relOut"                to "Fork 2" "release
        bind "Diner 2" "reqRght" to "L2" "reqIn"
        bind "L2" "reqOut"                to "Fork 1" "listen"
        bind "Diner 2" "getRght" to "L2" "getOut"
        bind "L2" "getIn"                 to "Fork 1" "grant"
        bind "Diner 2" "relRght" to "L2" "relIn"
        bind "L2" "relOut"                to "Fork 1" "release"
        bind "Diner 3" "reqLeft"          to "L5" "reqIn"
        bind "L5" "reqOut"                to "Fork 3" "listen"
        bind "Diner 3" "getLeft" to "L5" "getOut"
        bind "L5" "getIn"                 to "Fork 3" "grant"
        bind "Diner 3" "relLeft" to "L5" "relIn"
        bind "L5" "relOut"                to "Fork 3" "release
        bind "Diner 3" "reqRght" to "L4" "reqIn"
        bind "L4" "reqOut"                to "Fork 2" "listen"
        bind "Diner 3" "getRght" to "L4" "getOut"
        bind "L4" "getIn"                 to "Fork 2" "grant"
        bind "Diner 3" "relRght" to "L4" "relIn"
        bind "L4" "relOut"                to "Fork 2" "release"
}
```

Figure 12: An instrumented POLYLITH system specification for the Dining Philosophers problem.

MINION is a general-purpose configuration management tool. It can be used to create UNIX[2] *makefiles* [Feld78] and visualize petri nets, as well as to edit and execute POLYLITH MIL specs. More specifically, MINION is a general-purpose connected graph editor and simulator. We needed such a tool in order to visualize many types of nested structures: MIL specs, makefiles, and symbol tables. Each system places constraints on the construction of these connected graphs. MINION can accept different "grammars" which specify the semantic actions appropriate for a particular model.

MINION was written in C and it uses the PROTEGE (PROTotype Extensible Graph Editor) library. The PROTEGE library routines are written in C and PostScript and the system currently runs only in Sun's Network extensible Window System (NeWS) [GoRA89]. It implements the general-purpose connected graph edit and display functions within the NeWS server. The MINION client specifies node properties, linking semantics, and implements other MINION viewport control functions (e.g., I/O, transparency). Other tools, such as the diner windows and fork processes in Figure 10, may connect to a MINION viewport at runtime provided that they have a context number. This information is provided in stub headers created by the POLYLITH tools for each module.

6 SUMMARY

MINION has provided us with an extensible environment upon which to experiment with software system designs. We can quickly prototype new implementations based on ideas from old systems. It has allowed us to explore many different constructions that are textually tedious and difficult.

Visual programming systems are especially valuable in problem areas that have direct physical analogs. They enable us with new perspectives and insights into complex problems like link editing in heterogeneous, distributed software systems. Distributed applications are comprised of connected but autonomous protection domains that can be represented graphically so that we can visualize boundaries crossings. Designers of software systems that model physical systems also benefit from visual approaches. The Dining Philosophers problem is one such example. Finally, errors in connectivity can be more easily detected visually with an "at a glance" attention to detail. Such systems can complement textual representations where detail is dealt with on a more precise scale.

[2]UNIX is a trademark of AT&T Bell Laboratories.

References

[PuCa89] Purtilo, J., and J. Callahan, Parse-Tree Annotations, *Communications of the ACM*, (December 1989), pp. 1467-1477.

[Dijk65] Dijkstra, E. W., Cooperating Sequential Processes, Technical Report EWD-123, Technological University, Eindhoven, the Netherlands, (1965).

[Feld78] Feldman, S., Make - a program for maintaining computer programs, Bell Laboratories, 1978.

[GoRA89] Gosling, J., D. Rosenthal, and M. Arden, **The NeWS Book**, Springer-Verlag, New York, 1989.

[Purt86] Purtilo, J., A software interconnection technology to support specification of computational environments, Doctoral Dissertation, University of Illinois Department of Computer Science Report UIUCDCS-R-86-1269, (October 1986).